Improving generalized inverted index lock wait times

A Borodin, S Mirvoda, S Porshnev and O Ponomareva

IRIT-RTF, Ural Federal University, Yekaterinburg, Russia

Abstract. Concurrent operations on tree like data structures is a cornerstone of any database system. Concurrent operations intended for improving read\write performance and usually implemented via some way of locking. Deadlock-free methods of concurrency control are known as tree locking protocols. These protocols provide basic operations(verbs) and algorithm (ways of operation invocations) for applying it to any tree-like data structure. These algorithms operate on data, managed by storage engine which are very different among RDBMS implementations. In this paper, we discuss tree locking protocol implementation for General inverted index (Gin) applied to multiversion concurrency control (MVCC) storage engine inside PostgreSQL RDBMS. After that we introduce improvements to locking protocol and provide usage statistics about evaluation of our improvement in very high load environment in one of the world's largest IT company.
Keywords: Relational databases; Tree data structures; Concurrency control

1. Introduction

Generalized inverted index (Gin) is special tree-like data structure used in PostgreSQL for improving performance on set data types, such as arrays. It abstracts well known inverted index data type in a way that index doesn't know which operation it improves. Specific data type can implement its own index method strategy [1].

In this concept gin is different form ordinary B-tree, which support only predefined comparison operators and similar to GiST [GIST].

Gin define 4 different built-in strategies for arrays it can accelerate [2]:

- overlap – two arrays overlaps in any greater than zero number of values,
- contains – every element of the right array exist in the left array (order ignored),
- is contained by – every element of the left array exist in the right array (order ignored),
- equal – array on the left equals array on the right (order matters).

A Gin index consists of a B-tree index constructed over key values, where each key is an element of some indexed items i.e. element of array, and where each tuple in a leaf page contains either a pointer to a B-tree over item pointers (posting tree), or a simple list of item pointers (posting list) if the list is small enough.

The primary aim for Gin in PostgreSQL is high performance full text search querying. But it can be used in any task which can be presented as an array of tokens. It worth noting that Gin stores and looks for keys, not item values, that's why it heavily used for high performance querying of text array or numeric array for example article keywords, email recipients etc.

A. Index Layout

The index consists of pages, each containing as many index tuples as space allows. Index tuple is a structure containing: value of the indexed attribute (key datum) and tuple identifier (TID). TID is a structure pointing to physical location of data on disk. It tells on which page within a table or index the tuple is located and where on that page. For every table row, corresponding index tuple should exist, even if another row with the same key value already has been indexed. This duplication can cause page split, for one or more pages of the index.

TID duplication on posting lists affecting page size can be mitigated with compression similar to bitmap packing, which allows them to fit in much smaller. Compression is used for both the lists stored in-line in entry tree items, and in posting tree leaf pages.

Index layout composed of internal (entry tree) and leaf pages (posting tree or posting list). Internal page layout is looks exactly like B-tree and has no important differences from B-tree. But leaf pages have different layout and, as mentioned above, depends on number of elements.

When number of elements is small enough, they are stored as simple sorted list of tuple identifiers (TID), called posting list (Figure 1).

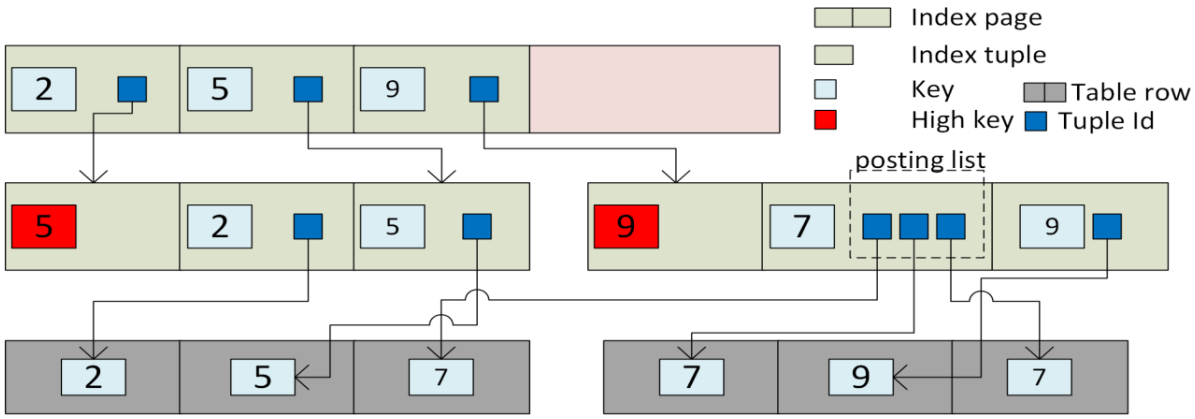

Figure 1. Posting list data structure.

When it is not, leaf pages contain a simplified B-tree called posting tree (Figure 2). High key is a special case of index tuple. It contains key value such that all tuples on the page are lower than or equal that value.

Layout features heavily used for managing concurrency issues, for example, we always certain about values of interest stored exactly righter and (or) deeper in the three hierarchy.

Another important aspect of design for concurrency shown in next section.

Posting list resides on the leaf page of entry tree. It contains ordered identifiers of heap tuples of data. If the posting list exceeds size of inline attribute (also called TOAST size, usually big attributes are stored in the oversized attribute storage - TOAST), then it is converted into posting tree.

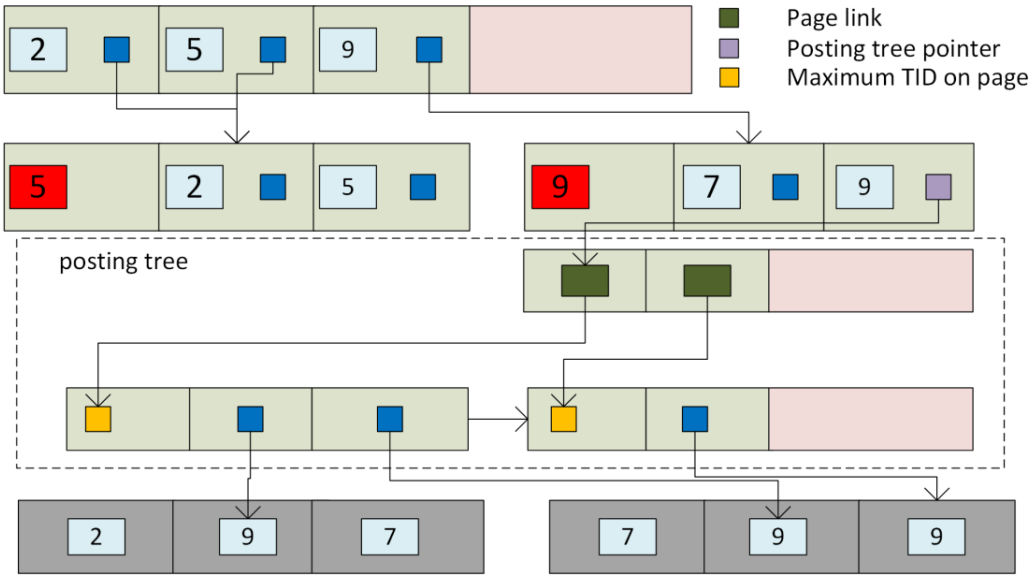

Figure 2. Posting tree data structure.

Key space for the entry tree is values of subelements of indexed attribute. Keys pace for posting list is identifiers of tuples in heap (TIDs). Keys pace for posting tree is, essentially, the same as the key space of posting lists. But posting tree is stored as B-tree.
Both posting trees and posting lists are ordered to make operation of intersection more efficient. E.g. if we scan an index for the query «green apple», we scan posting list of «apple» and posting tree of «green», and find every TID of tuple mentioning both «green» and «apple». This operation is performed for O(A+B) in worst case where A and B is size of posting trees or lists. In average complexity of the search in posting tree is O(R+H·log B), where R is the size of resulting tuples stream, H is height of B-tree and B is number of tuples in a block (page).

2. Concurrent architecture

As noted earlier the entry tree and each posting tree is a B-tree with right-links connecting sibling pages at the same level known also as B-link Tree [3].
To guarantee tree properties preservation during concurrent operations data structure have to obey so-called tree locking protocol, see [4, 5].
Numerous efficient locking protocols are described in [6]. It is worth noting that there are two kinds of locking objects: locks and latches. The first one is heavyweight operating system feature, the second one is lightweight language level construct or programming technique.
In PostgreSQL Gin tree locking protocol implemented with number of peculiarities which we show further.

a. Deadlock freedom

To avoid deadlocks, pages must always be locked in the same order: left to right, and bottom to top.

b. Searching

When searching, the tree is traversed from top to bottom, so the lock on the parent page must be released before descending to the next level. Concurrent page splits move the key space to right, so after following a downlink, the page actually containing the key we're looking for might be somewhere to the right of the page we landed on. In that case, we follow the right-links until we find the page we're looking for.

c. Deletion

To delete a page, the page's left sibling, the target page, and its parent, are locked in that order, and the page is marked as deleted. However, a concurrent search might already have read a pointer to the page, and might be just about to follow it. A page can be reached via the right-link of its left sibling, or via its downlink in the parent.

d. Insertion/Page split

A search descending the tree must release the lock on the parent page before locking the child, or it could deadlock with a concurrent split of the child page; a page split locks the parent, while already holding a lock on the child page. However, posting trees are only fully searched from left to right, starting from the leftmost leaf.

According to the design of Postgres [1] concurrent data editing implemented as snapshot isolation known as MVCC (Multiversion Concurrency Control) which guarantee that readers never block writers and writers never block readers [7].

Snapshot isolation lead to "dead" (not visible to any living transaction) tuples multiplication over time and degradation of index performance (when Postgres resolves TID provided by index, it can be "dead").

To overcome this problem from time to time Postgres execute special VACUUM service aimed to delete "dead" tuples and rearrange index pages.

Naïve implementation of can be as simple as:

```
foreach data_tuple
   check visibility
   if false
   drop data_tuple
foreach index
   drop index
   create index
```

At first glance, it looks like ordinate write activity, but actually it can lead to whole index and data rewrite, which is not viable for current volumes of data.

That's why vacuuming implemented differently for each one of the native PostgreSQL index (btree, gist, gin, hash) to provide best characteristics, achievable for specific data structure.

In the following section, we cover problem and solution of vacuuming in case of the general inverted index.

3. Problem statement

Vacuum of posting tree is doing two passes through posting tree:

1. Function ginVacuumPostingTreeLeaves() takes LockBufferForCleanup, effectively excluding all inserts. Then it traverses down trough tree taking exclusive lock, effectively excluding all reads. On leaf level it calls ginVacuumPostingTreeLeaf() function, which deletes all dead tuples. If there are any empty page, root lock is not released, it passes to stage two.
2. If there are any empty pages, ginScanToDelete() function scans through tree, deleting empty lead pages, then deleting empty inner pages, if they emerged after leaf page deletion. Between the above VACUUM invokes vacuum_delay_point(), which can hand for a while, holding CleanupLock on root leaf.

These lead to long lock wait times even for middle sized data load: for 50 GB data and 1 GB Gin index size exclusive write lock holds for 90 seconds or more, preventing any reads and writes from data.
Proposed and implemented solution changes function ginVacuumPostingTreeLeaves() such a way that now this it doing same depth first search as pass 1, but without cleanup lock, acquiring only read locks on inner pages and exclusive lock on leafs before eliminating dead tuples. If it finds empty leafs, it computes minimal subtree, containing only empty pages and start scan for empty pages from parent page pointing to found subtree. This scan acquires cleanup lock on root of scan (not necessarily root of posting tree).
Cleanup lock ensures no inserts are inside subtree. Scan traverse subtree depth first taking exclusive locks from left to right. For any page being deleted all leftmost pages were locked and unlocked before. New reads cannot enter subtree, all old read scans were excluded by lock\unlock.
Thus there shall not be deadlocks with ginStepRight() function. We get lock on page being deleted, then on a left page. ginStepRight() takes lock on left page, than on right page. But it's presence is excluded by cleanup lock and DFS scan with locks of upper and left parts of tree.
These modifications completely solve problem with vacuuming middle sized data, but can be improved even more.
The previous paragraph's reasoning only applies to searches, and only to posting trees.

4. Analisys

To protect from inserters following a downlink to a deleted page, vacuum simply locks out all concurrent insertions to the posting tree, by holding a super-exclusive lock on the parent page of subtree with deletable pages.

Inserters hold a pin on the root page, but searches do not, so while new searches cannot begin while root page is locked, any already-in-progress scans can continue concurrently with vacuum in corresponding subtree of posting tree. To exclude interference with readers, vacuum takes exclusive locks in a depth-first scan in left-to-right order of page tuples.

Leftmost page is never deleted. Thus, before deleting any page we obtain exclusive lock on any left page, effectively excluding deadlock with any reader, despite taking parent lock before current and left lock after current.

We take left lock not for a concurrency reasons, but rather in need to mark page dirty. In the entry tree, we never delete pages.

This is quite different from the mechanism the B-tree implementation uses to make page-deletions safe; it stamps the deleted pages with an XID and keeps the deleted pages around with the right-link intact until all concurrent scans have finished.

5. Future work

There are numerous ways to improve current approach. It's possible to rebuild locking protocol and B-tree algorithm in GIN to provide possibility of merging almost-empty pages as described in [4]. But the simplest yet not very clear technique to apply is a posting tree truncation. If the whole posting tree is empty, then we could mark root page as leaf and remove all other pages in tree without any locking.

Currently, during first posting tree scan we are detecting situation when root page shall be deleted, and prevent it, leaving leftmost pages of vacuumed tree. Posting tree height is never reduced. But in this situation, we could just replace the whole posting tree with the new empty root page. The concurrency protocol of this approach is not obvious: there may be still searches scanning through the tree, it's not possible to safely wipe pages.

This trick was suggested by PostgreSQL committers, but, after discussion we left it for future patches. From the performance point of view, this can be beneficial idea. Both, performance of VACUUM and

performance of scans. But doing so, we risk to leave some garbage pages in case of a crash. And it is not clear how to avoid these without unlinking pages one by one.

To implement, debug and evaluate described changes of the indexing, we successfully used techniques covered in our earlier works on this subject [8, 9].

6. Acknowledgment

The results were obtained with the financial support of the Russian Scientific Fund, Agreement N17-00-00000.

References

[1] Stonebraker M and Rowe L A (1986) The design of Postgres *Proceeding SIGMOD '86 Proceedings of the 1986 ACM SIGMOD international conference on Management of data* Vol 15 No 2 pp 340–355
[2] Interfacing Extensions To Indexes https://www.postgresql.org/docs/current/static/xindex.html
[3] Lehman P L and Yao S B 1981 Efficient Locking for Concurrent Operations on B-Trees *ACM Transactions on Database Systems* 6(4) pp 650–670
[4] Lanin V and Shasha D 1990 *Tree Locking on Changing Trees* (New York: New York University)
[5] Silberschatz A, and Z Kedem 1980 Consistency in hierarchical database systems *Journal of the ACM* Vol 27 pp 72–80.
[6] Graefe G 2010 A survey of B-Tree Locking Techniques *ACM Transactions on Database Systems* vol 35 16.
[7] Momjian B 2013 *Mvcc unmasked, unpublis hed* https://momjian.us/main/writings/pgsql/mvcc.pdf
[8] Borodin A Mirvoda S Kulikov I and Porshnev S 2017 Optimization of Memory Operations in Generalized Search Trees of PostgreSQL *International Conference: Beyond Databases, Architectures and Structures* (Springer, Cham) pp 224–232
[9] Borodin A Mirvoda S and Porshnev S 2015 Database Index Debug Techniques: A Case Study *International Conference: Beyond Databases, Architectures and Structures* (Springer International Publishing) pp 648–658